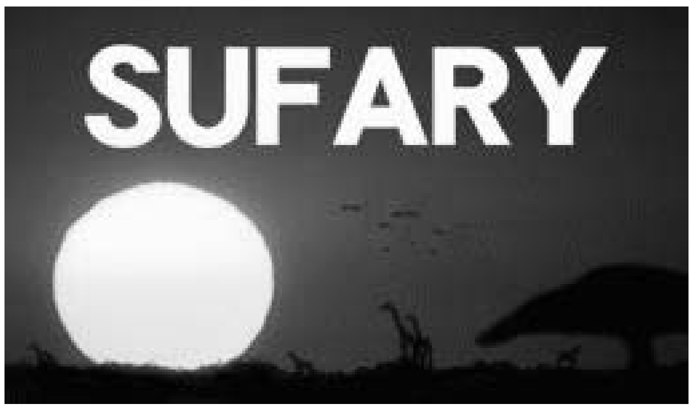
SUFARYガイド

山下達雄
tatuo-y@cl.aist-nara.ac.jp
奈良先端大学院大学 自然言語処理学講座

1999-06-29 版



本文書は SUFARYVersion 2.1 に準拠しています。この文書に関する御意見・御感想等ございましたら、sufary-adm@cl.aist-nara.ac.jp まで御連絡下さい。SUFARY の最新情報に関しては SUFARY ホームページ (http://cl.aist-nara.ac.jp/lab/nlt/ss/) を御覧下さい。

目 次



1　はじめに

SUFARY は、suffix array[1] というデータ構造を用いて高速な文字列検索を行なうためのライブラリを中心としたパッケージです。

SUFARY を使えば、事前にインデクックスを作成しておくことによって、grep のような手軽な全文検索を非常に高速に行うことができます。また、記事インデックスを作成しておけば、記事単位の検索も高速に行えます。

SUFARY で文字列検索を行うためのインデックスは array ファイル (suffix array) と呼ばれています。この array ファイルだけでは検索はできません。検索対象となるテキストファイルと array ファイルがそろって、始めて検索が可能になります[1]。記事単位の検索 (文字列検索結果をもとに記事を取り出す) を高速に行うために用いるインデックスは DocID ファイルと呼ばれています[2]。DocID ファイルには記事の開始位置などの情報が入っています。DocID ファイルが無くても記事単位の検索は可能ですが、検索速度は遅くなります。

SUFARY の得手不得手は以下の通りです。

得意なこと

辞書・住所録・新聞一年分データなどの、巨大な単一ファイルに対する高速文字列検索。

苦手なこと

電子メールのような多数のファイルに対する検索。WWW ページ検索などのファイルが手元にない状態での検索。

SUFARY ライブラリは、自作プログラムの検索部分などに高速検索機能を手軽に組み込めるように設計されています。また、Perl モジュールも用意されていますので、CGI 等での検索システムの構築も容易です。標準アプリケーションとして簡単な検索プログラム `sass` と対話型本格的検索プログラム `array` を用意しました。`array` には Tcl/Tk[4] による GUI もあります[3]。

第 2 章では、SUFARY のチュートリアルということで一通りの使い方について解説します。

suffix array による検索のしくみについては、第 3 章で詳しく解説します。他の検索手法との比較も行います。

第 4 章では検索のためのインデックスである array ファイルの構造と応用について、第 5 章ではテキスト領域 (記事) 検索のために必要な DocID ファイルの構造と応用について解説します。

---

[1] array ファイルを作成するアプリケーションとして mkary を用意しています。

[2] DocID ファイルを作成するアプリケーションとして mkdid を用意しています。記事の区切りを表す文字列を指定して、DocID ファイルを作成します。

[3] SUFARYversion 1.0[5] から 2.0 への変更点はかなりあります (特に `array` のコマンドなど)。大きな変更は、ライブラリの整備、アプリケーションの追加、複数ファイル対応、near 検索機能などです。ご注意下さい。

2　使ってみよう —チュートリアル—

ここでは、array ファイルの作り方、文字列検索のやり方、DocID ファイルの作り方、記事単位検索ののやり方について、実例を用いて一通り解説します。

検索用ファイル	使用目的	作成用プログラム
array ファイル	SUFARY での検索に必須	`mkary`
DocID ファイル	テキストエリア検索	`mkdid`

2.1　部分文字列を検索しよう

まずは array ファイルを作ってみましょう。`mkary` というプログラムを用います。

どういう単位で検索をしたいかという要求によって、作成される array ファイルが違ってきます。例えば、文字単位で array ファイルを作ると、そのテキストに含まれる全ての部分文字列が検索できます。samp1.txt に対して文字単位で array ファイルを作ると、“YAMASITA” や “Tatuo” はもちろん、“ASITA T” や “st-na” といった、半端な (?) 文字列でも見つけることができます。

samp1.txt

```
YAMASITA Tatuo
tatuo-y@cl.aist-nara.ac.jp
http://cl.aist-nara.ac.jp/~tatuo-y/
```

行単位で array ファイルを作ると、各行頭から始まる全ての文字列 (prefix といいます) が検索できます。samp1.txt に対して行単位で array ファイルを作ると、“YAMASITA”, “YAM”, “http” は見つかりますが、“aist” や “Tatuo” は行頭から始まる文字列ではないので、見つかりません。行単位の array ファイルは samp2.txt のような辞書の検索に向いています。array ファイルのサイズも文字単位のときよりも小さくなります。

samp2.txt

```
fish 魚
boy 男の子
girl 女の子
```

では、これらのファイルに対して array ファイルを作ってみましょう。作られた array ファイルの名前はデフォルトではテキストファイル名 ＋ “.ary” になります。

文字単位で array ファイルを作成

```
% mkary samp1.txt
Save to "samp1.txt.ary"
Reading text file "samp1.txt.ary"

Sorting...
Saving...
Done.
```

行単位で array ファイルを作成（`-l` オプション）

```
% mkary -l samp2.txt
Save to "samp2.txt.ary"
Reading text file "samp2.txt.ary"

Sorting...
Saving...
Done.
```

mkary のコマンドラインオプションについてはパッケージ付属の「検索用ファイル作成ガイド (MakeIndex.txt)」をごらん下さい。

検索結果を行単位で表示する簡単な検索プログラム sass を使って検索してみましょう。

簡単な検索の例

```
% sass 'nara' samp1.txt
15:16:tatuo-y@cl.aist-nara.ac.jp
42:15:http://cl.aist-nara.ac.jp/~tatuo-y/
% sass 'Tatuo' samp1.txt
0:9:YAMASITA Tatuo
% sass 'girl' samp2.txt
19:0:girl 女の子
% sass 'boy' samp2.txt
8:0:boy 男の子
```

sass 以外にも検索アプリケーションとして、対話型の array や Tcl/Tk[4] ベースの kwicview などがあります。詳しくは「付属アプリケーション利用ガイド (Tool.txt)」をごらん下さい。また、C ライブラリや Perl モジュールも用意してあります。これらについては「ReferenceC.txt」「ReferenceP.txt」をごらん下さい。

2.2 記事単位で検索しよう

さて、samp3.txt のような複数の記事 (<ARTICLE>タグと</ARTICLE>タグで囲まれているテキストエリア) が含まれるテキストがあるとします。「文字列『自然言語処理』を含む記事を見つけて取り出す」という検索を行うことを考えます。

samp3.txt

```
<ARTICLE>
形態素システム『茶筌』は２０世紀末に奈良先端大で開発された …（略）
 … フリーソフトとして公開 …（略）…
</ARTICLE>
<ARTICLE>
２１世紀初頭の自然言語処理システム開発への過剰な投資により、粗悪製品が
乱造され …（略）… 若者の自然言語処理ばなれが深刻 …（略）・
・・ 結局我々人間は歴史から何も学んでいないということを実感させられる。
</ARTICLE>
```

「自然言語処理」という文字列がどこにあるかは、array ファイルがあれば見つけることができますが、それがどの記事に含まれているかということは、array ファイルだけでは簡単には分かりません。そこで、SUFARY では DocID ファイルというファイルを利用します。DocID ファイルには記事の開始タグと終了タグ (この場合は<ARTICLE></ARTICLE>) の位置データが格納されていて、これによりテキストエリア検索が効率的に行えます。詳しいことは、第 5 章を御参照下さい。

ではさっそく DocID ファイルを作ってみましょう。DocID ファイルを作るプログラムは `mkdid` です。まず、なにはともあれ、array ファイルが必要なので、`mkary` で作ります。

```
% mkary samp3.txt
Save to "samp3.txt.ary"
Reading text file "samp3.txt"

Sorting...
Saving...
Done.
```

検索対象テキスト領域 (記事など) を囲むタグを引数に指定して、DocID ファイルを作ります。デフォルトでは DocID ファイルは samp3.txt.did という名前になります。

```
% mkdid '<ARTICLE>' '</ARTICLE>' samp3.txt
Number of Documents = 2
sorting...
writting...
done.
```

しかし、世の中、テキスト領域の始まりだけにしかタグがないというフォーマットも多いのも事実です。例えば以下のようなテキスト。

samp4.txt

```
#ID-001
形態素システム『茶筌』は２０世紀末に奈良先端大で開発された …（略）
 … フリーソフトとして公開 …（略）…
#ID-002
２１世紀初頭の自然言語処理システム開発への過剰な投資により、粗悪製品が
乱造され …（略）… 結局我々人間は歴史から何も学んでいないという
ことを実感させられる。
#ID-003
裏自然言語処理研究会のお知らせ：本日午後３時 …（略）… ふるって
御参加下さい。
```

こんなときは引数にタグを一つだけ指定すれば ok です。

```
% mkdid '#ID-' samp4.txt
Number of Documents = 3
sorting...
writting...
done.
```

テキストエリア検索を行う簡単なプログラム af を使って検索してみましょう。

検索の例

```
% af ’自然言語処理’ samp3.txt samp3.txt.did
FOUND 1
<ARTICLE>
２１世紀初頭の自然言語処理システム開発への過剰な投資により、粗悪製品が
乱造され …（略）… 若者の自然言語処理ばなれが深刻 …（略）·
·· 結局我々人間は歴史から何も学んでいないということを実感させられる。
</ARTICLE>
```

以上、基本的な使い方を手早く説明してみました。詳しいことは、詳細ドキュメントや後の章をごらん下さい。

3　検索のしくみ

SUFARY では suffix array というデータ構造を用いて高速文字列検索を実現しています。本章では、その suffix array による検索手法とその特徴を説明し、比較のために suffix array 以外の検索手法についても簡単に解説します。

3.1　suffix array について

当り前の話で恐縮ですが、本からある単語を探すには、巻末の索引を用いると便利です。辞書順 (50 音順、アルファベット順) に並んでいるので、簡単にお目当ての単語を見つけることができますし、指定されたページへ飛べば用は足ります。suffix array[1] は、電子化テキストに対してこの索引のような働きをするデータ構造です。ページの代わりにテキスト中の何文字目かを表す数字を用いています。しかし、本の索引のような見出しは無く、それだけでは単なる数字の羅列です。ページ番号だけの索引と思って頂ければ結構です。そんなもので検索ができるのかと疑われるかも知れませんが、それができるのです。後程、解説します。

さて、suffix array の仕組みについて解説します。suffix とは、ある位置から始まってテキストの終わりまで続く文字列を表すものだと考えて下さい。この suffix の始まりの位置を表す数字を suffix 番号 (あるいは、単に「番号」) と呼ぶことにします。「さくさくさくら」というテキストを例にテキストと suffix 番号の関係を下の図に示します。例えば、suffix 番号 3 の suffix は「さくさくら」、suffix 番号 6 の suffix は「くら」となります。

テキスト	さ	く	さ	く	さ	く	ら
suffix 番号	1	2	3	4	5	6	7

suffix array というのは、この suffix を表す番号の配列です。これらの番号は「番号が指す suffix を辞書順にソートした結果」に基づいて順番に格納されています[4]。言葉だけでは分かりづらいと思うので図 1 を御覧下さい。suffix「くさくさくら」は他の suffix と比べると、辞書順では一番先頭になります。ゆえに、suffix array の先頭には「くさくさくら」の suffix 番号 2 が格納されます。また、suffix「くさくら」は辞書順では先頭から 2 番目で、suffix 番号は 4 なので、suffix array の 2 番目に 4 が格納されます。

このようにして作成された suffix array を用い、二分探索 (binary search) で文字列検索を行ないます。例えば、図 1 で作成された suffix array を用いて、「くら」という文字列をキーワードとして検索してみます。二分探索では検索範囲の真中の要素とキーワードを比較して検索範囲を狭めて行きます。図 2 に検索の過程を示しました。まず、suffix array の真中である suffix 番号 1 に対応する suffix「さくさくさくら」とキーワード「くら」を比較します (図 2(1))。辞書順で考えると「くら」が小さいので、suffix array の真中より前半分に検索対象が絞られます (図 2(2))。次は前半分の中心 suffix 番号 4 の suffix とキーワードを比較します。今度は「くら」の方が大きいので後半分に検索対象が絞られます (図 2(3))。このように毎回検索範囲を半分に狭めていきます。この場合は suffix 番号 6 の suffix が最終的な検索結果となります。

当然のことながら検索結果が複数になることがあります。例えばキーワード「さく」での検索結果は suffix 番号 1 と 3 と 5 です。このように、検索結果が複数になるときは検索結果は suffix

[4] 文献 [7] の suffix array の定義を引用しておきます。

Definition　Giving an m-character string T, a *suffix array* for T, called Pos, is an array of the integers in the range 1 to m, specifying the lexicographic order of the m suffixes of string T.

array 上で連続 (連接) していることが分かります (1, 3, 5 は suffix array の 4 番目から 6 番目)[5]。というわけで SUFARY 内部では「suffix array の範囲」を検索結果として扱っています。「さく」の検索結果は「4〜6」といった感じです。

3.2 他の検索手法との比較

高速全文検索の様々な手法については文献 [6] で分かり易く解説されていますが、ここでは SUFARY との比較のために suffix array 以外の検索手法についても簡単に解説しておきます。

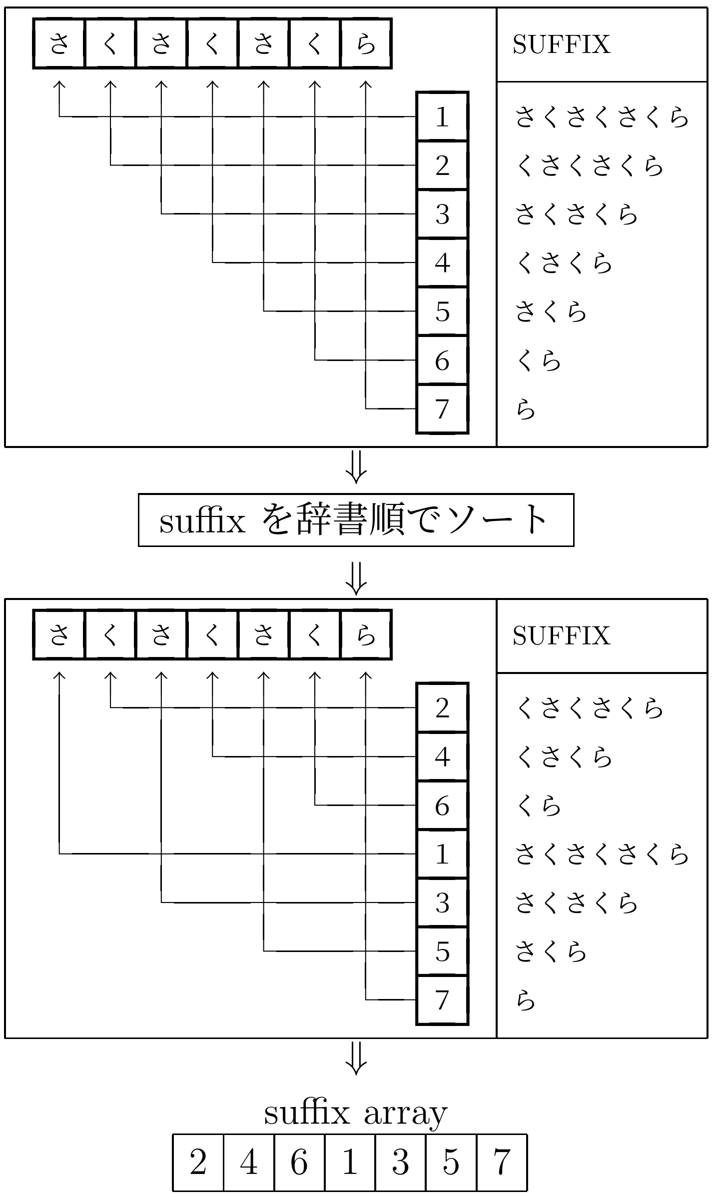


図 1: suffix array の作り方

[5]まあ、辞書順ですから当り前のことなんですけど。

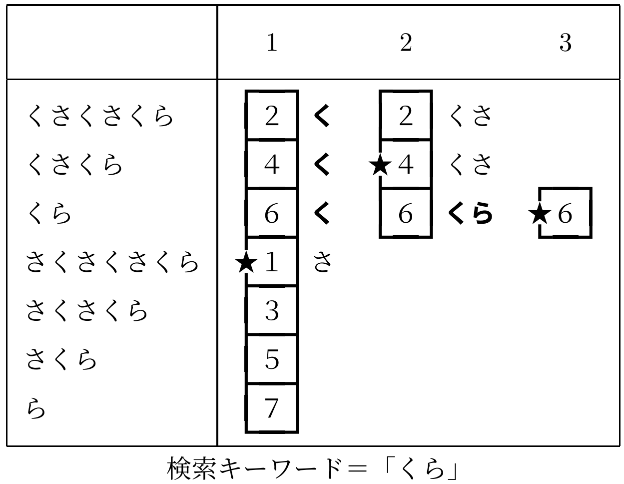


図 2: suffix array を用いた検索

様々な文字列検索手法を索引 (インデックス) という観点から分類してみました。

- インデックスを利用しないもの ………………… grep、Perl のパターンマッチングなど
- インデックスを利用するもの
  - テキストファイルが不要なもの …………… 部分列インデックス、シグネチャ など
  - テキストファイルが必要なもの ………………… パトリシア木、suffix array など

3.2.1 インデックスを利用しない検索

インデックスを利用しない検索は基本的にキーワードマッチングをテキストの最初から最後まで位置をずらしながら行ないます。一回に一文字ずつずらしながらマッチングを行なうのは大変なので、いかに大きく位置をずらすかというのが工夫のしどころです。様々なアルゴリズム (KMP, BM など) が考案されていますが [9]、結局、テキストファイルの大きさに比例した時間がかかってしまうので、検索対象が大規模なデータの場合は不向きです。

grep や Perl[10] の文字列検索がこれに当ります。

3.2.2 転置インデックス

インデックスを利用しない検索に対してインデックスを利用する検索は、インデックスの作成に時間がかかるものの、検索速度はテキストファイルの大きさに比例しないので高速な検索が行なえます。代表的なものは、本などの巻末の索引です。この「本の索引」をコンピュータ上で素直に再現したのが、**転置インデックス**[6]などと呼ばれる手法です。図 3 のように、文字列や単語を見出しとして、いわゆる「本の索引」(インデックスファイル) を作ります。インデックスファイルには見出しとページ番号のかわりの文書番号などが保存されています。

検索は簡単です。

[6] いろんな呼び方・方法がありますが、やっていることはだいたい同じです。

	…	11	12	13	14	15	16	17	18	19	…
文書 No.1	…	奈	良	先	端	技	術	大	学	院	…
文書 No.2	…	日	本	人	と	は	何	な	の	か	…
文書 No.3	…	奈	良	の	午	後	の	天	気	は	…

⇓

1-gram (一文字)

奈	(1:11) (3:11) ...
の	(2:18) (3:13) (3:16) ...
は	(2:15) (3:19) ...
ら	(3:18) ...
⋮	⋮

bigram(二文字)

奈良	(1:11) (3:11) ...
良先	(1:12) ...
先端	(1:13) ...
は何	(2:15) ...
⋮	⋮

単語

先端科学	(1:13) ...
日本人	(2:11) ...
大学院	(1:17) ...
奈良	(1:11) (3:11) ...
⋮	⋮

数字は文書番号と文書内での出現位置

図 3: 索引の素直な作り方

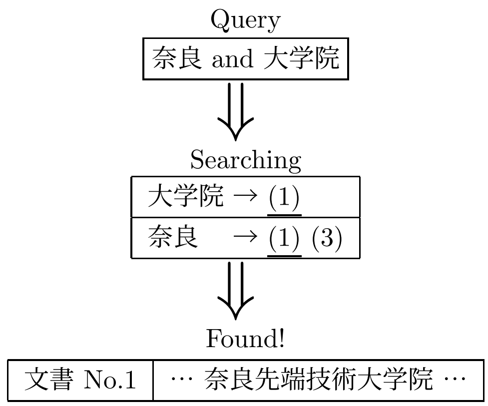


図 4: 転置インデックスによる検索の例

例 1 図 3 の単語インデックス (右の表) を用いた検索の例を図 4 にあげておきます。「奈良」と「大学院」が含まれる文書を検索しています。この例では、文書を探すだけなので文字位置情報は利用していません。「奈良」は文書 1,2 に「大学院」は文書 1 に含まれることがインデックスから分かるので、両方含まれる文書 (検索結果) は 文書 1 となります。

例 2 文字位置情報を用いた例として、図 3 の二文字インデックス (真中の表) だけを用いて、キーワード「奈良先端」を検索してみます。キーワードを二文字ずつ「奈良」「良先」「先端」に分解して、それぞれを表から探します。すると、文書 1 には (文書内での出現位置情報より) 3 つ全てが並んで出現しているのがわかり、検索結果は文書 1 ということになります。

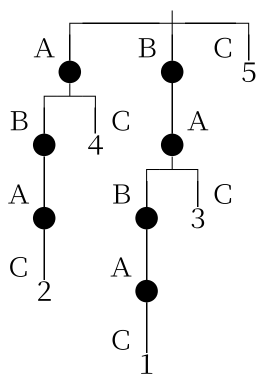
suffix array では、「『本の索引』の見出しの部分の文字列は、本文にもまったく同じ文字列があるんだから無駄だ。」という倹約思想に基づいます。おかげでインデックスファイルが簡略化でき**資源の節約**になります。しかし、検索の際にテキストファイル (本文) をいちいち参照しなければなりません。索引だけで検索ができる前述の方法とは本質的に異なります。WWW 検索 (キーワードによるホームページの検索) などでは、検索対象データ (つまり、あらゆる場所にある WWW ページデータ) を手元に持たない場合が多いので、部分列インデックスがよく使われます。suffix array

はこのような用途には向いていません。WWW 上で利用するのならば、同じサイトにあるデータの検索などに限定されます。

3.2.3 木

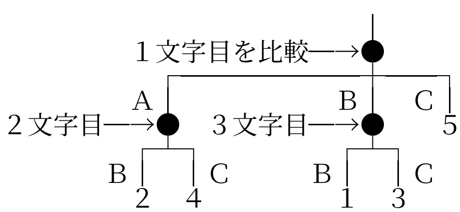
木構造 (トライ) でインデックスデータを構成する方法もあります。図 5 に例を示します。検索対象テキスト BABAC の全ての suffix で作成されたトライです (suffix tree[[7]] と呼ばれています)。終端ノードにはテキストへのインデックスが格納されています。非常に高速にキーワード検索が行なえます。しかし、大規模なテキストに対してはインデックスデータが巨大になり、あまり実用的ではありません。パトリシア木 [9] は、このトライを圧縮した構造で、テキストファイルを参照することにより、資源を節約しています[7]。suffix array の倹約ぶりにはかないませんが、suffix array の二分探索よりも高速であることは確かです。図 6 にパトリシアの例を示します。内部ノードには、比較を行う文字位置を記憶しておきます。この比較位置に従い、散発的に文字列比較を行うだけで済むので、終端ノードへ速くたどりつけるのです[8] 。

なお、suffix array で疑似的なトライを構成することもできます [11]。



テキスト	B	A	B	A	C
インデックス	1	2	3	4	5

図 5: トライ



テキスト	B	A	B	A	C
インデックス	1	2	3	4	5

図 6: パトリシア

[7]検索に用いるキーワードの長さを制限すれば、必ずしもテキストファイルが必要というわけではないのですが、あらゆる長さの部分文字列を検索できるようにするためにはインデックスデータが莫大になり過ぎるので、やはりテキストファイルを参照するようにするのが現実的です。

[8]ただし、正確を期すためには、最終的に suffix とキーワードの文字列比較をする必要があるます。

3.2.4 まとめ

全ての部分文字列 (あらかじめ決められたものではない**任意の文字列**) に対して検索ができるという条件で、様々な検索手法を比較してみます (転置インデックスは様々な実装方法があるので、簡単には比較できません) 。

検索方法	インデックスファイル	検索速度
インデックスを作らない	いらない	超遅い
suffix array	小さい	速い
トライ	大きい	suffix array より速い
転置インデックス	?	?

suffix array やトライはキーワード (あらかじめ決められたものではない任意の文字列) のテキスト中での出現位置を調べたりする厳密な文字列検索に有用です。しかし、転置インデックスは任意の文字列を確実に高速に検索するのには向いていません。とはいうものの日常の検索では、あらゆる文字列を検索する必要はほとんどないかもしれません。「東海道新幹線」で検索する人はいても、「道新幹」で検索する人はあまりいませんよね。結局、厳密さ、速度、インデックスのサイズのトレードオフになります。

SUFARY で suffix array を選択した理由は[9]、インデックスファイルも小さいし検索もそこそこ速く、コストパフォーマンスも申し分ないし、なによりインデックスの構造がシンプルだからです。

★ SUFARY の使われ方

当研究室は「自然言語処理学講座」ということもあって、「日本語の形態素解析 (文を名詞とか動詞とかに分割する処理) のための辞書検索」や「大規模なテキストデータから得た統計情報を用いて日本語解析を行なう際の統計情報の検索」などに利用されています。このような大規模な外部データはいちいちプログラム内部に格納して使用するより、SUFARY を使った方が簡単で効率的です。辞書検索的な使い方が多いですが、数百メガ規模の新聞テキストからの言葉の使用例の検索、ローカルニュースの検索、蔵書/論文の検索などテキスト検索にも利用されています。

辞書検索と言えば、部分列インデックス方式の検索エンジンなどのインデックスファイルは辞書とみなせます。というわけで、他の検索手法による検索エンジンのインデックスファイルの検索なんかに利用したりもしています。

[9] suffix array を選択したから SUFARY なんじゃないの、などというつっこみはなしね。

4 array ファイル

4.1 ファイルの構造

第 1 章で解説した通り、suffix array とはテキスト中の様々な位置から始まる半無限文字列 (suffix) の開始位置 (を表す数字) からなる配列で、それらの suffix を辞書順にソートして作られものです。

文献 [2] に非常にすっきりした suffix array の作成プログラム (図 7) が掲載されています[10]。SUFARY の suffix array 作成プログラム `mkary` はこれを参考に作りました。

SUFARY では、suffix の開始位置を表すのに long int を使っています。`mkary` で作成される array ファイルの構造は単なる long int の配列です。それゆえ、必要となるバイト数やバイトオーダーは array ファイルを作成した環境に依存することになるので注意が必要です。

array ファイルの中身を覗いてみましょう。テキスト「zenzendame」を例として array ファイルを作り UNIX の od コマンドで見てみます。

```
% echo -n zenzendame > samp_file
% mkary -q samp_file
% od -x samp_file.ary
0000000 0000 0007 0000 0006 0000 0009 0000 0004
0000020 0000 0001 0000 0008 0000 0005 0000 0002
0000040 0000 0003 0000 0000
0000050
```

実行環境は SunOS 5.5.1 ですので、long int は 4 バイト、バイトオーダーはビッグエンディアン (Big Endian) になっています。array ファイルの中身を見てみれば、以下のようにきちんとソートされているのが分かります。

```
#include <fcntl.h>
#include <malloc.h>
#include <stdio.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
/* usage: sufsort1 text > text.suf */

char *text;

suffix_compare(int *a, int *b)
{
  return strcmp(text + *a, text + *b);
}

main(int ac, char **av)
{
  struct stat stat_buf;
  int N, i, *suf;
  FILE *fd = fopen(av[1], "r");
  fstat(fileno(fd), &stat_buf);
  N = stat_buf.st_size;
  text = (char *)malloc(N+1);
  fread(text, sizeof(char), N, fd);
  text[N] = 0; /* pad with null */

  suf = (int *)malloc(N * sizeof(int));
  for(i=0;i<N;i++) suf[i] = i;

  qsort(suf, N, sizeof(int), suffix_compare);
  fwrite(suf, N, sizeof(int), stdout);
}
```

図 7: sufsort1.c

[10] `mman.h` を include する意図が私には分かりませんでした。

位置	SUFFIX
7	ame
6	dame
9	e
4	endame
1	enzendame
8	me
5	ndame
2	nzendame
3	zendame
0	zezendame

4.2 二段階の処理

mkary は、「array ファイルの書き出し」「array ファイルのソート」という二段階の処理を行います。

mkary のオプション No Sort (-ns) は、「array ファイルの書き出し」処理だけを行いソートを行わないというものです。つまり、そのままでは検索に利用できない不完全な array ファイルを作成します。以下の実行例を見れば分かるように、suffix を表す位置情報が順番に並んでいるだけです。どんな使い道があるのかはあえて明言しません。

```
% mkary -q -ns samp_file
% od -x samp_file.ary
0000000 0000 0000 0000 0001 0000 0002 0000 0003
0000020 0000 0004 0000 0005 0000 0006 0000 0007
0000040 0000 0008 0000 0009
0000050
```

No Sort の双子の兄弟みたいなオプションが Sort Only (-so) です。これは、すでに存在している array ファイルに対して「array ファイルのソート」処理のみを行うというものです。以下の実行例に示すように、array ファイル (samp_file.ary) が存在しないとソートのしようがないのでエラーになります。No Sort で作成した後なら実行できます。結果は普通に実行したときと同じにきちんとソートされます。

```
% echo -n zenzendame > samp_file
% mkary -q -so samp_file
ファイル "samp_file.ary" がオープン出来ません。
% mkary -q -ns samp_file
% mkary -q -so samp_file
% od -x samp_file.ary
0000000 0000 0007 0000 0006 0000 0009 0000 0004
0000020 0000 0001 0000 0008 0000 0005 0000 0002
0000040 0000 0003 0000 0000
0000050
```

Sort Only 機能にはいろいろな使い道があります。まず一つ目はインデックスの追加です。テキストファイルの末尾に新たなテキストを追加する場合、もういちど最初から array ファイルを作り直すのが普通です。しかし、テキストファイルが巨大な場合、作り直す時間をなるべく短縮したいものです。「array ファイルの書き出し」「array ファイルのソート」という二段階の処理を行いま

す。そのような場合はテキストファイルにテキストを追加するだけでなく、array ファイルにも位置情報を追加して Sort Only 機能を利用するのが現実的です。この流れを以下の例でごらん下さい。あくまで例なので、テキストファイルは巨大なものではありませんが。

```
% echo -n zenzendame > samp_file
% mkary -q samp_file
% echo -n 'jan' >> samp_file   ### テキストファイルへの追加
% perl -e 'print pack("lll",10,11,12);' >> samp_file.ary   ### array
 ファイルへの追加
% od -x samp_file.ary   ### 0..0a, 0..0b, 0..0c が末尾に追加された
0000000 0000 0007 0000 0006 0000 0009 0000 0004
0000020 0000 0001 0000 0008 0000 0005 0000 0002
0000040 0000 0003 0000 0000 0000 000a 0000 000b
0000060 0000 000c
0000064
% mkary -q -so samp_file
% od -x samp_file.ary
0000000 0000 0007 0000 000b 0000 0006 0000 0009
0000020 0000 0004 0000 0001 0000 000a 0000 0008
0000040 0000 000c 0000 0005 0000 0002 0000 0003
0000060 0000 0000
0000064
```

最終的には以下のようにきちんとソートされているのが分かります。

位置	SUFFIX
7	amejan
11	an
6	damejan
9	ejan
4	endamejan
1	enzendamejan
10	jan
8	mejan
12	n
5	ndamejan
2	nzendamejan
3	zendamejan
0	zezendamejan

二つ目は自由なインデキシングです。`mkary` ではオプションの指定により文字単位 (`-c`)、単語単位[11](`-w`)、行単位 (`-l`) によるインデキシングができます。しかし「形態素解析の結果を用いて形態素単位にインデキシングしたい」とか「新聞記事のキーワードが列挙されている部分だけでインデキシングしたい」とか「辞書データの見出し語の部分だけでインデキシングしたい」など様々な要求があります。`mkary` でこのようなことを全て可能にするのは非常に大変です[12]。そこで、まず、これらのインデキシング基準で、ソートされていない array ファイルを作り、そして、その array ファイルを Sort Only 機能でソートするという手順が現実的です。この流れを以下の例でごらん下さい。母音が現れる位置でインデキシングしています。

[11] スペース等のデリミタで区切られた単位
[12] インターフェース設計や実装が面倒臭いという意味です

```
% echo -n zenzendame > samp_file
% perl -ne 'while(/(.)/gs){print pack("l",pos()-1) if($1 =~ /[aeiou]/)}'
 samp_file > samp_file.ary
% od -x samp_file.ary   ### aeiou が現れる位置の配列
0000000 0000 0001 0000 0004 0000 0007 0000 0009
0000020
% mkary -q -so samp_file
% od -x samp_file.ary
0000000 0000 0007 0000 0009 0000 0004 0000 0001
0000020
```

最終的には以下のようにきちんとインデキシングされてソートされているのが分かります。

位置	SUFFIX
7	ame
9	e
4	endame
1	enzendame

4.3　分割とマージ

`mkary` で suffix array を構築 (array ファイルをソート) するとき、テキストと array がメモリ上にのっていれば非常に望ましい環境です。メモリが少いと仮想記憶のページングが多くおこり非効率的になります。ページングの数を押えるために array ファイルを分割し、局所的にソートを行い、最後にまとめる (マージ) という方法をとることができます。

この方法をとる場合は、`-b` オプションで array ファイルの分割数を指定します。以下に使用例 (5 ブロックに分割) を示します。

```
% mkary -b 5 sample
Save to "sample.ary"
```

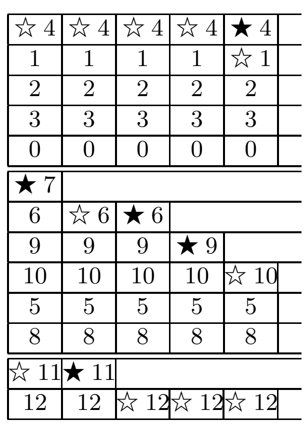
(1)
3 つのブロックに分割

位置	SUFFIX
0	zezendamejan
1	enzendamejan
2	nzendamejan
3	zendamejan
4	endamejan
5	ndamejan
6	damejan
7	amejan
8	mejan
9	ejan
10	jan
11	an
12	n

(2)
各ブロック内でソート

位置	SUFFIX
4	endamejan
1	enzendamejan
2	nzendamejan
3	zendamejan
0	zezendamejan
7	amejan
6	damejan
9	ejan
10	jan
5	ndamejan
8	mejan
11	an
12	n

(3)
マージ



位置	SUFFIX
7	amejan
11	an
6	damejan
9	ejan
4	endamejan
1	enzendamejan
10	jan
8	mejan
12	n
5	ndamejan
2	nzendamejan
3	zendamejan
0	zezendamejan

図 8: 分割&マージの例

```
Reading text file "sample"
++
Sorting...
 BLOCK 1
 BLOCK 2
 BLOCK 3
 BLOCK 4
 BLOCK 5
Merging into "sample.ary-"
Rename "sample.ary-" with "sample.ary"
Done.
```

3 つのブロックに分割したときの処理の流れを図 8 に示します。まず、ソート前の array ファイルを 3 つのブロックに分割します。そして、各ブロック内でソートを行います。局所的な処理なのでページングの回数が押えられます。最後に各ブロックの先頭から順番に一つずつ suffix をとりだし、それらを比較してその中で辞書順に最小なものを選択して別のファイルに出力していきます。例えば図 8 の (2) では最初に 4, 7, 11 の suffix の中から辞書順で最小である 7 が選ばれ別ファイルに出力されます。suffix 7 は 2 つ目のブロックの先頭だったので、次の (2 つ目のブロックの) 先頭 suffix は 6 になります。今度は、4, 6, 11 の suffix の中から辞書順で最小である 11 が選ばれ別ファイルに出力されます。この次は 4, 6, 12 の中から 6 が、その次は 4, 9, 12 の中から 9 が選択されます。このように 3 つのブロックを辞書順を保ったままマージします。各フェーズで辞書順で最小の suffix を選択するときにはヒープ [9] というデータ構造を用いていますので、各選択フェーズの計算時間は $O(\log \text{ブロック数})$ しかかかりません。マージ時にはページングが多くおこりますが、分割せずにソートするよりかは少いので、結果的に速度は向上します。

4.4 ソート速度の向上

文献 [12] に suffix array を高速に構築するさまざまな方法が紹介、及び、提案されています。単純な Quick Sort ではなく、文字列のソート (suffix のソート) に特化した方法をとることで高速化が可能になります。

5　DocID ファイル

suffix array を用いて文字列検索を行うと、その文字列のテキスト中での位置 (何文字目) は即座に分かるのだが、それがどのテキスト領域 (記事など) に含まれているかを調べるには別の手法が必要になる。

図 9 を例に考えてみる。array ファイルがあれば、「フリーソフト」という文字列が、テキストの 630 文字目にあるということはすぐ分かる。しかし、それが「形態素解析システム『茶筌』の歴史」という記事に含まれているということは 630 という位置情報 (suffix 番号) だけでは分からない。

テキスト中の対象となるテキスト領域の開始位置と終了位置を交互に昇順に並べた配列 (DocID 配列 = DocID ファイル) を利用する。図 10 に図 9 のテキストを対象にした DocID 配列の例を示す。文字列検索結果が含まれているテキスト領域を知るには、文字列検索結果である suffix 番号を検索キーとして、その値を越えない最大の開始位置をその配列から二分探索で探せば良い。

例 1　文字列「フリーソフト」が 630 文字目にあるとする。 配列を二分探索して 630 を越えない最大の数字を探すと 472。 472 は記事の開始位置なので、求める記事は 472 文字目から 790 文字目まで[13] のテキスト領域ということが分かる。

例 2　文字列「間だよん」が 804 文字目にあるとする。 配列を二分探索して 804 を越えない最大の

位置 (suffix 番号)	テキスト
	···
472：記事開始位置	<DOC><TITLE>形態素解析システム『茶筌』の歴史</TITLE>
	<CONTENT> ···
630	フリーソフトとして公開 ··· </CONTENT></DOC>
790：記事終了位置	
790	記事と記事との
804	間だよん。
814：記事開始位置	<DOC><TITLE>自然言語処理バブルの崩壊</TITLE>
	<CONTENT> ··· </CONTENT></DOC>
1326：記事終了位置	
	···

図 9: タグ付けされたテキスト領域を持つテキストの例

記事 (DOC タグで囲まれているテキスト領域) の DocID 配列

13	210	472	790	814	1326	1406	1763	1840	2199
開始	終了	開始	終了	開始	終了	開始	終了	開始	終了

タイトル (TITLE タグで囲まれているテキスト領域) の DocID 配列

···	486	533	819	859	···
	開始	終了	開始	終了	

図 10: DocID 配列の例

[13] DocID の隣の要素を見れば終了位置が分かる

数字を探すと 790。790 は記事の終了位置なので、「間だよん」を含む記事はないということが分かる。

開始位置と終了位置が交互に現れるという構造なので、開始位置か終了位置かは瞬時に判断可である。付加情報は不要。

この DocID 配列は記事だけでなく階層構造を持つテキストの検索に利用できる。例えば、あるキーワードが含まれているタイトル (`<TITLE>` と `</TITLE>` で囲まれている) を探し、その記事 (タイトルを含んでいる) を取り出したいという場合、図 10 のようなタイトルの DocID 配列も用意すれば良い。

例 3 タイトル (`TITLE`) に文字列「バブルの崩壊」が含まれる記事 (`DOC`) を探す。

文字列「バブルの崩壊」が 838 文字目にあるとする。838 を越えない最大の数字を タイトルの DocID 配列から探すと 819。819 はタイトルの開始位置なので、「バブルの崩壊」を含むタイトルは 819 から始まるということが分かる。このタイトルの開始位置 819 を越えない最大の数字を記事の DocID 配列から探すと 814。求める記事は 814 文字目から 1326 文字目のテキスト領域である。

このように、多階層の構造をもつ文書にたいする様々な階層での検索が可能になる。これは嬉しい。XML とかに応用可能です。その場合は、データディクショナリのようなテキスト構造やファイル構成を管理する凝った枠組が必要になるでしょう。

DocID 配列は `mkdid` によって作成され、対象テキストファイル名 ＋ “.did” というファイル (DocID ファイル) にセーブされます。構造は array ファイルと同様で、単なる long int の配列です。UNIX の `od -x` コマンドなどでのぞいてみましょう。

6 SUFARY の歴史

そもそもことの始まりは、1996 年の春から夏にかけての、奈良先端科学技術大学院大学のプロジェクト実習という授業の課題「パトリシア木 [9] による高速文字列検索プログラムの作成[14]」であった。

自然言語処理学講座の松本裕治教授が単位認定者、当時 M2 の山下達雄 (松本研) が現場監督にあたった。当初は、松本研内で改良中であった形態素解析器 JUMAN(後に「茶筌」となる) の辞書引き部分の高速化を目標としていた。山下の作成したプロトタイプを元に、当時 M1 の熊谷俊高 (????研)、米沢恵司 (松本研) が作業にあたった。

その後、このプロジェクトは、数百メガバイト規模のテキストデータの高速検索を目指したが、パトリシア木のような木構造はデータサイズが膨大になり、当時の (現在も?) 松本研のマシンでは太刀打ちできないことが判明した。そこで、効率的なデータ構造でかつ高速検索の可能な suffix array[1] を採用することとなった。熊谷が検索、コマンドインターフェースに関する部分を作成し、山下がその他の周辺部分の作成、および、修正・保守・整理を行なった。

それから、約一年後、1997 年 6 月に、suffix array を用いた高速文字列検索プログラムは山下により勝手に SUFARY と命名され暫定版が公開された。7 月には正式版 (Version 1.0)[5] が公開され、ホームページも開設された。

山下により細々と保守されていた SUFARY であるが、1997 年夏から、松本研 M2(当時) 今村友明の開発参加により、複数ファイル処理、near 検索などの機能が追加された。また、松本研 D2 中山拓也 (当時) により、Perl モジュールが作成され、(松本研内の) 様々な研究に利用された。

1998 年初夏、検索部分の C ライブラリ化が完了し Version 2.0 が公開された。

1998 年秋、山下によりテキスト領域 (記事など) 単位で検索できるよう機能拡張された。

(敬称略)

[14] 正確な題は忘れた。

7　お知らせ

SUFARY に関する情報は以下の URL を御参照下さい。バグやリリースなどの最新情報も入手できます。また、SUFARY に関する技術情報もときどき提供しています。

http://cl.aist-nara.ac.jp/lab/nlt/ss/

参考文献

[1] Udi Manber, Gene Myers. “Suffix Arrays: A New Method for On-line String Searches”, 1st ACM-SIAM Symposium on Discrete Algorithms, pp.319-327, 1990.

suffix array の原典。

[2] Kenneth W. Church. “You shall know a word by the company it keeps”, NLPRS’95: Natural Language Proceeding Pacific Rim Symposium, p.22-34, 1995.

[3] 長尾真編. “自然言語処理”, p.20-23, 岩波書店, 1996.

[4] John K. Ousterhout(著), 西山芳幸・石曽根信 (訳). “Tcl & Tk ツールキット”, ソフトバンク株式会社, 1995.

[5] 山下達雄, 熊谷俊高, 米沢恵司, 松本裕治. “Suffix Array を用いた高速文字列検索システム SUFARY Version 1.00 取扱説明書”, SUFARY パッケージ Version 1.00 付属, July 1997.

[6] 道本健二, 真島馨. “高速全文検索の威力”, 日経バイト, 1996 年 10 月号.

[7] Dan Gusfield. “Algorithms on Strings, Trees, and Sequences”, Cambridge Univ. Press, 1997.

DNA 配列の検索を目的とした検索アルゴリズムに関する本です。suffix array に関する詳細な解説が載っています。原典より分かりやすいです。

[8] Gaston H. Gonnet, Ricardo A. Baeza-Yates, Tim Snider. “New Indices for Text: Pat Trees and Pat Arrays”, in “Information Retrieval”, pp.66-82, Prentice Hall, New Jersey, 1992.

OED(Oxford English Dictionary) のコンピュータ編集のために、patricia 木 [9] から suffix array (Pat array) を作成する手法が提案されています。我々も patricia 木から suffix array を作成するプログラムを実装してみましたが、大規模なデータに対してはそもそも patricia 木が作成できないので (計算機環境の問題) 断念しました。

[9] Robert Sedgewick(著), 野下浩平・星守・佐藤創・田口東 (訳). “アルゴリズム (Algorithms)”, 近代科学社，1992.

「原書第 2 版 第 2 巻 探索・文字列・計算幾何」の patricia 木の解説は非常に分かり易いです。

[10] Larry Wall, Tom Christiansen, Randal L. SchWartz 著, 近藤嘉雪 訳. “プログラミング Perl”, オライリー・ジャパン, November 1998.　原題: “Programming Perl *Second Edition*”

[11] 山下達雄, 松本裕治. "Suffix Array を用いたフルテキスト類似用例検索", 情報処理学会研究報告 97-NL-121, pp.83-90, September 1997.

[12] 伊藤秀夫. "大規模テキストに対する Suffix Array の効率的な構築法", 情報処理学会研究報告 99-NL-129, pp.27-34, January 1999.